

AD-A167 886

**AIDING COMPUTER APPLICATION PROGRAMMERS AND USERS WITH
THE TOOLS OF THE VISUAL INTERFACE(U) NAVAL POSTGRADUATE
SCHOOL MONTEREY CA M N FREDERICKSEN MAR 86 1/1**

UNCLASSIFIED

F/G 9/2

NL

541
177



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

2

AD-A167 886

NAVAL POSTGRADUATE SCHOOL

Monterey, California



DTIC
ELECTE
MAY 27 1986
S D D

THESIS

AIDING COMPUTER APPLICATION PROGRAMMERS
AND USERS WITH THE TOOLS OF THE
VISUAL INTERFACE

by

Michael Neils Fredericksen

March 1986

Thesis Advisor:

Gordon H. Bradley

Approved for public release; distribution is unlimited.

DTIC FILE COPY

60 11 11 8

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

AD-A167886

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED			1b. RESTRICTIVE MARKINGS		
2a. SECURITY CLASSIFICATION AUTHORITY			3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.		
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE			5. MONITORING ORGANIZATION REPORT NUMBER(S)		
4. PERFORMING ORGANIZATION REPORT NUMBER(S)			7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School		
6a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School		6b. OFFICE SYMBOL (if applicable) Code 52	7b. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000		
6c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000		9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER			
8a. NAME OF FUNDING/SPONSORING ORGANIZATION		8b. OFFICE SYMBOL (if applicable)	10. SOURCE OF FUNDING NUMBERS		
8c. ADDRESS (City, State, and ZIP Code)		PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.
11. TITLE (Include Security Classification) (U) AIDING COMPUTER APPLICATION PROGRAMMERS AND USERS WITH THE TOOLS OF THE VISUAL INTERFACE					
12. PERSONAL AUTHOR(S) Fredericksen, Michael N.					
13a. TYPE OF REPORT Master's Thesis		13b. TIME COVERED FROM TO	14. DATE OF REPORT (Year, Month, Day) 1986 March		15. PAGE COUNT 61
16. SUPPLEMENTARY NOTATION					
17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)		
FIELD	GROUP	SUB-GROUP	User Interface, Visual Interface, Visual Tools, Toolbox, Application Programmers		
19. ABSTRACT (Continue on reverse if necessary and identify by block number) The purpose of this thesis is to explicate the benefits of the computer visual interface by identifying the functional capabilities of some of the visual devices made possible by such an interface, and by examining the ways in which these visual devices can provide tools to aid the programmer in writing, debugging, and unit testing application programs, and the user in learning and using applications. This thesis should give the reader some insight into how current visual technology is being used (or can be used in the future) to aid the applications programmer and application user. The focus will primarily be on the programmers and users of applications for low-cost desktop computers.					
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT <input type="checkbox"/> DTIC USERS			21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED		
22a. NAME OF RESPONSIBLE INDIVIDUAL G. H. Bradley			22b. TELEPHONE (Include Area Code) (408) 646-2359	22c. OFFICE SYMBOL Code 52Bz	

Approved for public release; distribution is unlimited

Aiding Computer Application Programmers And Users
With The Tools Of The Visual Interface

by

Michael N. Fredericksen
Lieutenant, United States Navy
B.B.A., James Madison University, 1978

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ENGINEERING SCIENCE

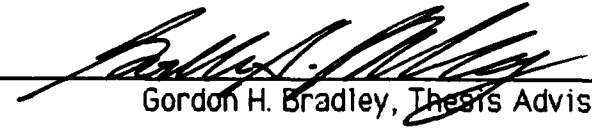
from the


NAVAL POSTGRADUATE SCHOOL
March, 1986

Author:

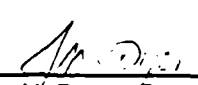

Michael N. Fredericksen

Approved by:


Gordon H. Bradley, Thesis Advisor


Daniel L. Davis, Second Reader


Vincent Lum, Chairman, Department of Computer Science


John N. Dyer, Dean of Science and Engineering

ABSTRACT

The purpose of this thesis is to explicate the benefits of the computer visual interface by identifying the functional capabilities of some of the visual devices made possible by such an interface, and by examining the ways in which these visual devices can provide tools to aid the programmer in writing, debugging, and unit testing application programs, and the user in learning and using applications. This thesis should give the reader some insight into how current visual technology is being used (or can be used in the future) to aid the applications programmer and application user. The focus will primarily be on the programmers and users of applications for low-cost desktop computers.

Accession For	
NTIS CRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution	
Availability Codes	
Dist	Avail and/or Special
A-1	



TABLE OF CONTENTS

I.	INTRODUCTION	5
II.	TOOLS OF THE VISUAL INTERFACE	10
	A. TERMINOLOGY	10
	B. A LIST OF VISUAL DEVICES	10
III.	VISUAL DEVICES TO AID THE APPLICATIONS PROGRAMMER . .	17
	A. THE PROGRAMMING TASK	17
	B. AIDS FOR WRITING AND EDITING THE PROGRAM	19
IV.	THE USER INTERFACE	34
	A. BACKGROUND	34
	B. THE CHANGING USER	34
	C. THE STANDARDIZED INTERFACE	36
	D. DISCUSSION	41
V.	CONCLUSION	58
	LIST OF REFERENCES	59
	INITIAL DISTRIBUTION LIST	60

I. INTRODUCTION

The introduction of computers with bit-mapped, high resolution screens during the 1970s spurred the development of a new medium for human--machine interaction: the visual interface. Initial research into the utilization of this interface was carried out on high-cost devices by universities and private concerns, most notably the Xerox Palo Alto Research Center (Xerox PARC).

When technological advances made it possible to implement a visual interface economically on a desktop machine in the late '70s, many of the research ideas generated at Xerox PARC were combined into a single, relatively low-cost system, the Apple Macintosh™. It combined a high-quality visual display, sufficient computer power, and, most importantly, a comprehensive, visually-oriented systems interface.

The Macintosh™ was the result of an effort to create a powerful, easy-to-use desktop machine which, utilizing a standardized visual interface, would decrease user learning time and ease transitions when going from one application to another. Recognizing the need for standardization, Apple incorporated a set of tools (a toolbox) for the creation of the many elements of the visual interface into the ROM of the machine, accessible to applications programmers via a special programming language.

The success of the Macintosh™ has stimulated great interest in visual interfaces and has paved the way for the development of similar systems: 1) machines that have *Macintosh™-like* interfaces (eg., the Commodore

Amiga™ and the Atari 520ST™); and 2) software products that create a Macintosh™-like visual interface on systems that use linguistically-oriented operating systems (eg. Digital Research's GEM™ and Microsoft's WINDOWS™).¹ One reason for much of the interest in visual interfaces lies in the changing nature of the computer user--the widespread use of computers in almost every field of human endeavor has created two new types of users: 1) the manager--user of many applications; and 2) the casual user--the infrequent user of applications. "Nonspecialists require a rich interactive environment making use of graphics and audio communication" [Ref. 1] in order to be productive--the visual interface provides this rich environment. Another reason for the increasing interest in the visual interface is the changing nature of application programs. The domain of the computer has extended beyond that of just wordprocessing, file management and number crunching--computers are being used with page-layout applications, CAD (Computer Aided Design) and engineering applications, and graphically oriented database programs--areas of use made possible by enhanced display capabilities.

The visual interface is having an extraordinary impact on the computer industry: 1) it is expanding the range of use of computers to areas previously ignored because of the need for high quality graphics; 2) it is broadening the user base to include managers, casual users, and those not knowledgeable in the ways of computers; and 3) it is facilitating the

¹ The reader should note that the term *Macintosh™-like* as used here refers to the visual similarities to the Macintosh™ interface and does not imply that it was intentionally emulated by other products.

development and use of software applications whose complexity would prohibit use on a non-visual interface.

A written explanation detailing the reasons for this impact is difficult to find since the development of the Macintosh™ interface and similar systems was accomplished with little or no conceptual explanation--they were just presented and sold as products. Hence, the only way a potential user can develop an understanding of the benefits of the visual interface (outside of this thesis) is to buy or borrow one of these products and use it. This fact, together with the misconceptions arising from a general lack of understanding of such an interface and the relatively high cost of visually oriented systems, has undoubtedly slowed the widespread acceptance of systems utilizing a visual interface.

The purpose of this thesis is to explicate the benefits of the computer visual interface by identifying the functional capabilities of some of the visual devices made possible by such an interface, and by examining the ways in which these visual devices can provide tools to aid the programmer in writing, debugging, and unit testing application programs, and the user in learning and using applications. This thesis should give the reader some insight into how current visual technology is being used (or can be used in the future) to aid the applications programmer and application user. The focus will primarily be on the programmers and users of applications for low-cost desktop computers.

The visual interfaces of the low-cost systems currently available are remarkably similar. We will use the process of abstraction to rise above the details of each system's particular implementation, create an abstract system interface, and then generate a list of the basic components

of this interface--the visual devices (and their respective capabilities) that make up the display that the user sees [Ref. 2]. This abstraction makes it possible to view each existing system as an example of the abstract system combined with specific implementation decisions.

The terminology used to refer to visual devices in this thesis refers to the visual devices of the *abstract* interface, rather than of any particular system. We shall normally (but not always) use the terminology of the Macintosh™ interface¹, since it appears to be the most descriptive. Apple was the first to introduce a low-cost visually-oriented system interface in the Macintosh™, and was not restricted by copyright laws in developing its terminology for the visual tools as were subsequent firms. We intend however that the terminology apply to the abstract system, and thus, need not be attributed to a specific system. Specific definitions of features or details of operation also refer to the features of the abstract system; hence, it is not necessary to attribute them to the system in which they appear.

The incorporation of a greatly enhanced display technology into low-cost machines has challenged programmers to find ways to exploit the new capabilities offered by bit-mapped, high-resolution displays. Some of

¹ This thesis is about the tools of the visual interface, therefore, it is important that we have a standard terminology for referring to these tools. Some of the terms which we have chosen to use are part of a terminology developed by Apple Computer, Inc. (Cupertino, CA) for their Macintosh™ computer, and are registered trademarks or are protected by copyright. This terminology includes the terms: Pointer, Select, Click, Double-Click, Mouse, Desktop, Window, Dialog Box, Desk Accessory, Icon, Toolbox and Pull-down Menu.

the capabilities and features described in this thesis are exactly as available on current systems; others are from research and development efforts on visual systems [Ref. 3]. This thesis presents a discussion of the union of these capabilities plus some additional capabilities which would be useful, but which are not yet implemented, that go along with the spirit of the interface. The thesis is organized into four parts, of which this introduction is the first. Part II introduces the reader to the visual devices and the capabilities they provide (together these form the visual tools). Part III discusses ways in which visual tools can be used to help the applications programmer; this is illustrated through the use of a hypothetical program (a program with which to write programs). Part IV of the thesis is devoted to the user interface--its creation by the applications programmer, and its use by the application user. Part V is the conclusion.

II. TOOLS OF THE VISUAL INTERFACE

A. TERMINOLOGY

The terms *administrative* and *non-administrative* are referred to frequently throughout this thesis. *Administrative* tasks are those operations that are activated from outside application programs. Examples of these tasks include: naming, combining, printing, copying and deleting files, and initializing, booting, ejecting, erasing, copying and naming disks. The user associates the accomplishment of these, and other functions dealing with whole files or whole disks, with the administrative environment. *Non-administrative* tasks are those tasks which are activated from within applications (they are application specific).

The term *background* is used (only in conjunction with a visual interface) to describe the whole-screen, graphical image that is furthest to the rear of the screen. All windows, icons, dialog boxes and other visual tools are projected on top of some background (a background cannot be manipulated, although it has features that can be used). A background is usually used as a frame of reference and is named in accordance with the function it performs: in the administrative environment, the background is referred to as the desktop; in the environment of an application, the background is given some name that is appropriate to its use.

B. A LIST OF VISUAL DEVICES

The following is a list of visual devices that provide system users with powerful tools which are easy to use and remember, and which allow

applications programmers to create an environment which varies little from application to application. The visual devices are described both in terms of their primitive functional aspects and in terms of the capabilities they provide when enhanced by software.

1. Pointer

The Pointer is a tool which gives the user a rapid means of pointing to any geographical location (pixel) on the screen. When facilitated through the use of a mouse, the Pointer can select individual pixels (by clicking), create linear arrangements of pixels (by dragging), and select predefined patterns of pixels (ie. icons, windows etc., by clicking). These basic functions make it possible for the Pointer to do the following:

- (a) Access pull-down menus and select command options.
- (b) Select choices from dialog boxes.
- (c) Select, move and resize icons and windows.
- (d) Select and manipulate screen areas containing graphics or text.
- (e) Select pixels linearly to create two-dimensional graphics.
- (f) Launch applications or open documents.

The Pointer may change form to suit different applications (i.e. it may be an arrow in the administrative environment, an insertion-point bracket in a word processing application, or a pair of scissors in a page layout application), but its principles of operation should remain the same--the user should not have to relearn its basic functions.

The Pointer is the user's primary tool for manipulating, creating, and editing screen elements, and for selecting actions or elements to be

acted upon. By not having to memorize keyboard commands, the user is free to concentrate on what he is doing with the information rather than how he is doing it.

2. Windows

Windows are organizational frames of reference for the user. They provide a means of organizing information graphically and a means of displaying "large amounts of information onscreen simultaneously" [Ref. 3]. When combined with the use of icons, windows provide a way of visualizing the actual location of an element of information (represented by an icon) and a basis for its manipulation. By having multiple windows onscreen simultaneously, the user can arrange his elements or aggregates of information as he pleases without having to go to a lower level of abstraction, and he gets immediate feedback on any actions that he takes in the form of highlighting or other screen changes. Windows also can provide a means of observing two or more processes simultaneously (for example, watching a graphics program execute line-by-line and observing its output as it is drawn in another window).

3. Dialog Boxes

Dialog boxes are the system's best means of communicating with the user. Often incorporating menus of options, dialog boxes force decisions to be made and give immediate (and specific) feedback (in the form of error messages or system failure messages) on problems with the system, application, or user inputs. Within applications, they provide prompts to the user and give him a selection of options and a means to select those options (for example, when quitting an application, the user might be presented with a dialog box inquiring whether the changes that have been made to the

document that he was working on should be saved or discarded--his subsequent selection either activates a save to disk process or discards the changes).

4. Pull-down Menus

Pull-down menus are expandable menus that can be viewed in their entirety by selecting the menu name and dragging downwards toward the bottom of the screen. The menu names appear across the top of any background screen. Pull-down menus perform several functions:

- (a) They give the user a list of what actions are available to him at any time (highlighted selections show what actions are possible now, while unhighlighted selections indicate that something is required from another process or the user before they become possible).
- (b) They guide the user down the decision path when he is uncertain of what to do next.
- (c) They give the user access to powerful processes without ever touching the keyboard (routines and functions can be activated by selecting a single menu item).
- (d) They are tailor-made to fit the environment of the system or application so that the user is always aware, by the menus available, of which environment he is in.
- (e) They facilitate much faster learning of the system and applications because the user is not required to memorize a command language (they are the command language), and thus necessitate far fewer references to manuals.

5. Desk Accessories

An extension of pull-down menus, these are powerful mini-programs that run in the background of an application or the desktop. They can be used without leaving the present environment.

The power of desk accessories lies in the ability they give the user to perform administrative tasks from a non-administrative environment, and in the fact that they provide the user with valuable tools which he can use within an application, even though they were not included in its software. In some ways, they are very much like library routines, except that they are called directly by the programmer or user and not by the application program, and are wielded by the programmer or user as tools, rather than used as subroutines.

6. Icons

Icons are graphical symbols used to represent meaningful elements or aggregates of information. Icons adhere to the "picture-is-worth-a-thousand-words" concept in that, to be useful, they must immediately convey their exact meaning to the user at a glance. They can represent simple identifiers for physical things such as disks or applications, or they can represent complex concepts such as files or relational objects (i.e. folders).

System icons are system created and are used to represent standard elements in the administrative environment (ie. disks, trash, folders, files). The user is usually permitted to name these icons with names that are designed to jog his memory as to exactly what they represent; thus, they provide a way for the user to tailor his visual interface to suit his particular way of thinking. Since system icons are components of the standardized visual interface, they remain the same regardless of the application, creating a familiar environment for the user.

Application icons, created by applications programmers, vary among applications, necessitating that the user learn the meanings of the icons within a specific application. This is usually not difficult because: 1) they are limited in number; and 2) an icon's form is designed by the application programmer to be visual representation of its function or contents. The use of application icons is a way of customizing the visual interface to fit a particular application, thus making it easier for the user to learn. It is important that the application programmer ensure that the rules for the manipulation of these icons remain the same as for the system icons, so that the user is not required to learn and memorize any new rules.

Some applications allow the application user to create icons which serve as identifiers to represent specific, user created functions or items of data to be manipulated or used within the application. This is particularly true of certain visually oriented database applications, where a user created icon might be used to represent a unique item for which the database contains underlying information.

Icons can be moved around easily with the mouse, enabling the user to move the large amounts of information that they may represent into an organized format that he is better able to work with--they allow him to see and manipulate the big picture.

7. Desktop

The desktop is the background of the administrative environment of the system. It provides a geographical frame of reference to the user and a framework in which to manipulate and organize (to his liking) lower-order objects (windows, icons etc.). When the system user is presented with the

desktop, he immediately knows that he is in the system (administrative) environment, and is aware of the tasks that can be accomplished there.

8. Controls

Controls are visual devices which emulate (with a few exceptions) physical devices--they are manipulated as if they were physical objects rather than graphical images. Controls can be used to: 1) activate processes--they can take the form of buttons in a dialog box or window which the user "pushes" (clicking with the mouse) to execute a particular command (e.g. a *Close box* on a window); 2) manipulate objects--they can take the form of *scroll bars* (for scrolling through the contents of the window) or *Expander boxes* (for increasing or decreasing the size of the window) on windows; and 3) regulate the features of certain other objects--for example, they can be used to modify certain system characteristics--increasing or decreasing the volume of the audio output, or the frequency of the cursor blink, or adjusting the responsiveness of the keyboard. Controls are designed so that their operation is physically obvious--their form describes the method by which they are operated.

III. VISUAL DEVICES TO AID THE APPLICATIONS PROGRAMMER

A. THE PROGRAMMING TASK

The actual writing of an application, as discussed in this thesis, involves the implementation (coding) of a specified algorithm. Devising an algorithm to solve a specific problem is here considered part of program design, and not part of the implementation of the program. Thus, the actual implementation of an application is the area which shall be focused upon. This task consists of devising a plan of attack (possibly several, using different methods and different data structures to accomplish the same task), coding the plan, testing the code, making modifications and corrections, retesting, and documenting.

The following is a discussion of the visual tools which could be used to aid the programmer in the writing, testing and debugging of application programs. Suggestions as to how these visual tools might be used are included.¹

1. Capabilities To Be Provided By Visual Tools

Before proceeding, it is important that we discuss the capabilities that we wish to provide the programmer through the use of visual tools. The following is a list of these capabilities :

¹The reader should note that since implementation is necessarily language dependent, these visual tools may not be applicable to all languages--we leave it to the reader to determine which ones are applicable to which languages.

- (a) Give the programmer graphically oriented word processing and editing tools to help him in the initial writing of the program.
- (b) Let the main portion of the programmer's efforts go toward the writing and refining of the problem solving features of the application by providing him access to tools which simplify the creation of the user interface (give him a toolbox with tools for creating windows, dialog boxes, pull-down menus, icons etc. with toolbox routines).
- (c) Enable the programmer to create hierarchical structures that make use of abstraction and zoom-in so that he is able to see the big picture, or the greatest detail (or any level in between) of the program on which he is working (compacting information).
- (d) Enable the programmer to step through the program to observe the effect of each line of code on the output of the program, the contents of the data structures, or on the value of variables.
- (e) Enable the programmer to execute commands at any point during program execution (after any line of code) which allow him to view parameter or variable values, or allow him to experiment with quick fixes without actually changing the program (using an Instant Command Window).
- (f) Enable the programmer to eliminate redundant or repetitive tasks by giving him sophisticated tools to edit and manipulate code and store his efforts for later use (libraries of icons, templates, and coded subroutines).
- (g) Free the programmer from having to worry about proper syntax and static semantics by having the system do that for him.
- (h) Give the programmer sophisticated search and editing devices which simplify the debugging and maintenance of the program.

The question should be asked at this point: What sort of visual tools provide these capabilities? Visual tools for programmers can be

classified into three categories: wordprocessing and editing type tools, organizational and search tools, and library building/tapping tools. Each of these categories makes its own contribution to improving programmer productivity (and program quality), but each does it in a distinctly different way.

B. AIDS FOR WRITING AND EDITING THE PROGRAM

The discussion of visual tools to aid the programmer in the writing and editing of an application program is most easily accomplished through the description of a *hypothetical program* that provides such tools.¹ It should be noted that the initial writing of the program does not lend itself to being aided by the use of visual devices, with the exception of sophisticated word processing tools and program libraries. Editing functions, and fast access to a library of reusable code and routines, provide some opportunity to increase programmer productivity. However, these increases are generally small, and thus, aside from offering a few tools which save the programmer a little typing, visual aids for the initial writing of the program are of limited value. We shall discuss the most valuable ones.

1. A Hypothetical Program

The program we require is specifically designed for writing and editing applications. It makes extensive use of windows. The main window

¹ The reader should note that the hypothetical program discussed in this thesis focuses on aiding programmers working with traditional imperative languages. A similar hypothetical program (not discussed in this thesis) could be produced to aid programmers in functional programming.

serves as a background for all other windows, and is at least partially visible at all times. This window contains the present version of the developing program, gives the programmer a geographical frame of reference to which he can return at any time, and is used for the initial writing of the program and the selection of elements for the edit windows. It is also used for compiling--this is the only window that the compiler can see, hence, any changes made elsewhere must be written to this window (Note: All the edit windows and applicable dialog boxes provide a feature to do this). The main program window allows either line by line or rapid scrolling through the program in either direction.

The program has a feature called Instant Info which allows the user to query the system about certain program elements. It works as follows:

- (a) Double clicking on a variable or constant displays a dialog box containing the variable name, its type, its initial value, and the names of the procedures and functions it is used by (see Figure 1).
- (b) Double clicking on a function name or procedure name displays a window containing its name, its callers, its callees, its required inputs, its returned outputs, its termination condition (if applicable), and its internal type, variable, and constant declarations (see Figure 2).
- (c) Selecting a variable or constant and then choosing CREATE EDIT WINDOW (these windows are discussed later) from the edit menu creates an edit window stack containing all occurrences of the variable or constant, and also highlights them in the main program window (see Figure 3).

INSTANT INFO	
VARIABLE OR CONSTANT NAME	TYPE (if applicable)
Head	PassPtr
SUBROUTINES IT IS USED BY (if any)	INIT VALUE/VALUE
prog FlightList proc ReadPass proc GetPass	
<div>RET</div> <div>CANC</div>	

Figure 1
The Instant Info Window For Variables Or Constants

SUBROUTINE NAME		INSTANT INFO		TERMINATION COND.	
proc GetPass				I=50	
CALL BY	PASSED	type	RETURNS	type	
prog FlightList	Head Next I	PassPtr PassPtr integer	NewPass I NewFlt	Passeng integer integer	
CALLS	PASSES	type	IS RETURNED	type	
ReadPass	Head	PassPtr	NewPass NewFlt	Passeng integer	
CountPass	I	integer	I	integer	
		VARIABLE	type	CONSTANT	value
LOCAL VARIABLES AND CONSTANTS		XPtr	Passptr	maxseats	50

Figure 2
The Instant Info Window For Subroutines

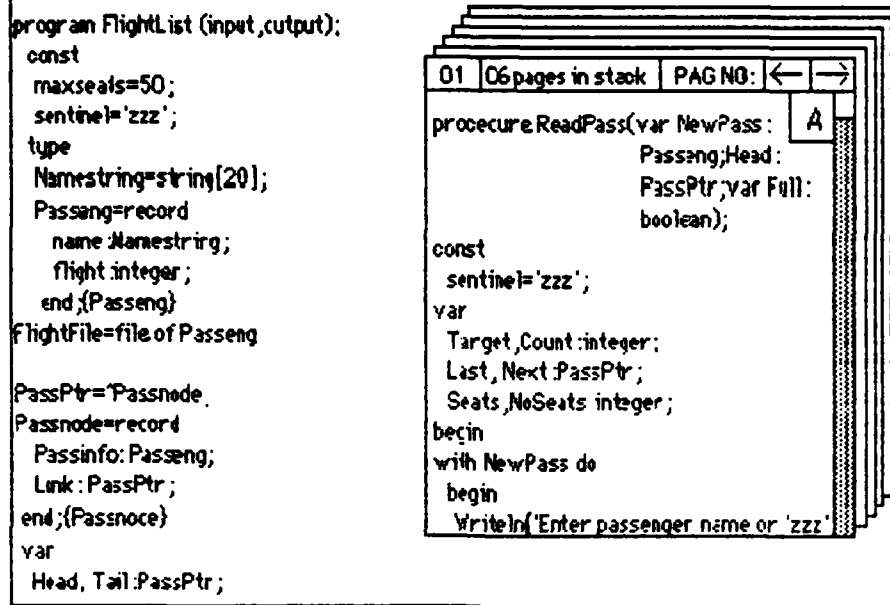


Figure 3
Edit Windows

a. Declarations

Most modern imperative programming languages require the programmer to declare the variables, constants, and types that are used within the program. Such declarations create an environment in which the program is to run. The writing of declarations can be tedious for large programs which use many variables, constants and types. Although it is conceivable that this process could be automated (see dialog box below), it appears that such automation would be no faster than manual declarations and might actually slow down the programmer. In view of this, it would seem that the traditional method of declaration for languages that require it remains the most efficient--there is an exception to this: while writing a program the programmer often finds himself having to create new variables,

the need for which was not previously foreseen. Two methods could be used for dealing with this situation:

- (a) A simple, automated method of declaring variables through the use of a dialog box (which could prove a real time-saver--it would relieve the programmer of having to scroll back to the beginning of the program or procedure.
- (b) A "rapid transit" mechanism which would allow the programmer to be instantly transported from one part of the program to another, using program, function, or procedure names as geographical keys.

(1) The Automated Method. In our hypothetical program, the automated method works as follows: the user utilizes the pointer to pull-down the PROGRAM menu (see Figure 4), chooses CREATE DECLARATIONS from it, is presented with a dialog box (see Figure 5), and then points to, selects and fills in the appropriate choices within the box. As each item is declared, a new dialog box appears for the next item until all declarations are made. When no more entries are desired, double clicking on the name block of the dialog box returns the programmer to the main window. The dialog box "CREATE DECLARATIONS" can be used for declaring global and local variables, types and constants for the main program and subroutines. To delineate which is chosen, the programmer selects (points and drags across) the program name or the appropriate function or procedure name and then chooses CREATE DECLARATIONS from the PROGRAM menu. This places him in the proper context.

PROGRAM (menu)
CREATE DECLARATIONS (choice)
 .
 .
 .
UNIT TEST (choice)
CREATE EDIT WINDOWS (choice)

Figure 4
The Program Menu

CREATE DECLARATIONS

NAME:

type	<input type="checkbox"/>	TYPE: real	<input type="checkbox"/>	INIT VALUE	<input type="text"/>
const	<input type="checkbox"/>	integer	<input type="checkbox"/>	VALUE	<input type="text"/>
var	<input type="checkbox"/>	boolean	<input type="checkbox"/>		
		char	<input type="checkbox"/>		
		string	<input type="checkbox"/>		
		array	<input type="checkbox"/>		
		enum.	<input type="checkbox"/>		

length

range

enum. list

Figure 5
The Create Declarations Dialog Box

(2) The "Rapid Transit" Mechanism. We will define a *Rapid Transit* mechanism as a device which allows the programmer to move swiftly from one part of the program to another. In our hypothetical

program, the utilization of such a mechanism is accomplished through the use of a window (we'll call this the RAPID TRANSIT WINDOW). An icon for summoning this window is a constant element of any screen in the main program window (it is always onscreen, in some convenient location, for ready access). Opening this icon, the user is presented with a window containing a comprehensive list of the program, procedure, and function names within his program. By selecting one of these names, the programmer is transported to the program location containing the declarations for that name (this is an example of zooming in on the details). A return feature is provided for returning the programmer to the original location, as well as other enhancements for stepping through the locations in both directions.

This Rapid Transit mechanism is useful for checking and adjusting program, procedure, and function parameters, as well as making new declarations. To illustrate, let's look at an example: if the programmer is writing a procedure call to Cancellist in program FlightList and cannot remember all the parameters needed to call Cancellist, he can use the Rapid Transit Window to transport himself to the declaration of Cancellist, get the information that he needs, and then transport himself back to the procedure call to Cancellist and continue coding. The figure below (Figure 6) shows what this window might look like. The procedure Cancellist is selected in the name list portion of the window. Double clicking on the name or selecting GO THERE NOW will transport the programmer to the declaration of the procedure; selecting the Return box will transport him back. If he desires to remain at the declaration of the procedure, he can select the Cancel Return box, which will close the RAPID TRANSIT WINDOW and cause the icon to reappear in the main program window. If he decides to go on to

another location, the computer will keep track of each location visited (in order), allowing him to back track or go forward to any location already visited.

Program, Procedure, or Function Name	Return:
prog FlightList	<div>RAPID TRANSIT WINDOW select the name of your destination</div> <div>GO THERE NOW</div> <div>PREVIOUS LOC</div> <div>NEXT LOCATION</div> <div>CANCEL</div>
proc ReadPass	
function CheckCount	
proc UpdateList	
proc CancelList	
	Cancel Return:

Figure 6
The Rapid Transit Window

b. Debugging

Program debugging offers perhaps the most significant opportunity to increase programmer productivity through the use of visual devices. This is because much of the time spent debugging is devoted to searching for and locating the cause of an error rather than actually correcting it. The visual interface simplifies this search process through the use of powerful visual tools. Let's examine some examples of such tools and the things that can be done with them.

In order to realize the full potential of visual tools in the area of debugging, it is necessary to use an interpreter during the debugging process. This is because many of these visual debugging tools require that the running program be stopped in mid-execution or executed line by line.

The process of debugging can involve the correction of many types of errors, and visual tools can be used in finding and correcting many of these. Nevertheless, in this thesis, we will focus on the visual debugging tools which bring about the greatest increases in programmer productivity.

(1) Syntax And Static Semantic Corrections. Let's start with syntax and Static semantic errors. With a non-visual interface, syntax and static semantic (e.g. type) errors are discovered at compile time (when using a compiler). The compiler generates error messages, usually referring to specific locations; the programmer writes down the error messages on paper, scrolls to the designated location, corrects the errors, and recompiles the program. Even given a good text processor that saves some scrolling, this process can be cumbersome.

It would be much more efficient to detect syntax errors as they are made and correct them on the spot. The use of a syntax directed editor [Ref. 4] during coding doing syntax checking (and formatting), combined with the visual effect of highlighting, makes this possible. As the programmer types in his lines of code, any syntax errors are highlighted as they are made (after the return key is pressed or after an end of line deliniation such as the semi-colon in Pascal). The programmer immediately knows he has made an error and corrects it before going on. This is a significant time saver.

(2) Program Monitoring. A much more effective time saver can be realized by enabling the programmer to step through the program line-by-line, and providing him with the capability of examining the value of variables and parameters as he goes. This is particularly effective because it deals with the search aspect of debugging--it allows him to watch the execution of the program in detail, monitor the changing values of his variables and parameters, and subsequently locate the cause of the problem when the execution fails to follow the expected path or the value of a variable changes to something unexpected (this tool is useful in locating program logic errors).

(a) The Instant Command And Observation Windows. The *Instant Command Window* and the *Observation Window* are visual devices which allow the programmer to monitor the values of variables, parameters and expressions--while the program is being executed. The Instant Command Window enables the programmer to execute instant commands at any point during the step-by-step execution of the program (e.g. `WriteLn(variable);` in Pascal). This tool allows the selection of specific points in the program for the examination of the values of variables, or the verification of parameter values that are passed from one subroutine to another. When using the Instant Command Window, it is important that the programmer have some means of stopping program execution at points which he designates, allowing him to pass quickly through the parts of the program that work properly and get to the location of the suspected error. This is accomplished by enabling the programmer to place "stop" icons at those points in the program where execution is to be stopped. In addition to allowing the programmer to check the values of his variables and

parameters, the Instant Command Window enables the programmer to insert "quick fix" commands into the program during execution and to observe their effect without actually changing the program.

Another window which allows the programmer to monitor the values of variables and parameters is the Observation Window. This window is especially effective for watching the values of variables or parameters during an iterative process. The programmer types in the expression he wants to monitor and executes the program--the values of the variable, parameter or expression are updated automatically after each iteration (the programmer doesn't have to give any commands to make this happen each time). This window also offers the advantage of not cluttering up the screen--unlike the Instant Window, it displays evaluation results itself and does not require any other window for its operation.

(b) The Output Window. Use of the Instant Command Window necessitates having an *Output Window* that allows the programmer to monitor program output as the program is being executed line-by-line. This window receives the same output that the programmer would normally receive after the program had run. It is useful in that the programmer can see the program's outputs while watching the step-by-step execution of the program in the main program window. The reader should note that the power of these visual tools lies in the fact that the programmer can watch different aspects of his program simultaneously, and thus, pinpoint errors in the program simply by noting the location--the specific step during execution--where the error occurred.

(3) Editing Windows. It is in the *editing windows* that most of the program modifications are made by the programmer. The number and

contents of these windows varies. The following methods are used to create them:

- (1) The programmer can select portions of the program at random which he desires to edit by selecting the applicable text and choosing COPY TO EDIT from the EDIT menu. This allows non-destructive manipulation of the code--he can work with the code segment in the editing window while leaving it intact in the main program window.
- (2) The programmer can activate a search for specified elements of the program. For instance, he can activate a search for all program portions where a certain variable or constant appears--the system searches for all occurrences of that variable or constant and copies the line on which the variable or constant appears plus the next eleven lines following it (or some number that is practical with regard to screen size) to the edit window. Each occurrence of the variable or constant is highlighted in the edit window. The user is also able to activate searches for all calls to a certain procedure or function in order to check for errors in parameter passing.
- (3) The programmer can work on a particular subroutine by selecting its name and then selecting CREATE EDIT WINDOW from the PROGRAM menu. The entire subroutine will be inserted into an editing window (see Figure 3).

When the programmer creates an editing window stack, a separate window is created for each selection that is copied to the stack. Selections that are greater than some specified length (determined by screen size) will enable the scroll bar on the edit window, allowing the user to scroll through the entire selection. All windows are numbered sequentially based on their contents' actual position in the program, not the order of selection. They appear as a stack of windows that can be paged through in either direction via a page bar located across the top of the window, which consists of two horizontal directional arrows for page

turning, a numerical indicator of the number of pages in the edit window stack, and a page number block which tells the user which page he is on.

Multiple editing window stacks (up to the maximum number that can be accommodated by the size of the screen) are able to be displayed onscreen simultaneously. The stacks can be named by the user or they will receive some specified default names, for example, A-E. They can be dragged around the screen and arranged as desired, but are restricted from the leftmost inch of the screen which always maintains the image of the main program window. Windows within a stack can be separated from the stack for window comparisons etc. This does not affect page turning.

(a) Type Checking: Using Edit Windows. Type errors are often difficult to trace down in languages that are not strongly typed, especially in a large program. A tool is needed that allows the programmer to specify the variable that he wants to type-check and then retrieves all the instances where that variable is used in the program or subroutine (depending upon whether it is local or global) for the programmer to examine for errors. In our hypothetical program, the user can select the variable to be type-checked by double clicking on it. This creates a stack of edit windows which contains all usages of the selected variable throughout its scope. In addition, this highlights all usages of the variable in the main program window. Examining these windows should reveal the source of the error, which can then be corrected directly in the edit window.

(4) Unit Testing: Using An Environment Generator. Unit testing is presently complicated by the necessity of writing complex test harnesses or program stubs in order to create the environment in which the program is to run. This is one of the reasons testing consumes such a large

portion of program development time. Significant increases in programmer productivity could be gained if the programmer were provided with an interactive *environment generator* that used dialog boxes or windows for communicating with the programmer. This device would allow the programmer to create the module interfaces (test harnesses and stubs) required for the testing of the module, and would allow him to manipulate the values of the parameters passed to the module in accordance with his test specification.¹ With such a device in mind, let's see how unit testing is done in our hypothetical program.

When he wants to test a subroutine, the programmer selects UNIT TEST from the PROGRAM pull-down menu and is presented with a dialog box where he has to fill in the blanks and answer simple computer generated queries regarding parameters needed as inputs, parameters used internally by the subroutine, data structures needed by the subroutine, and control information to be accepted and/or passed by the subroutine. The dialog box is actually the communications link between the programmer and the underlying environment generator. The latter is a sophisticated device that performs the functions of making declarations, initializations, typing,

¹ The reader should note that the implementation of an environment generator is extremely complicated and is not feasible for some languages. Those languages where it is possible vary as to how much environment lends itself to generation and how much must be borrowed from the main program. Although it would be desirable to be able to create complete testing environments through the use of such a device, a true, whole environment

and the construction of data structures. The programmer supplies the necessary information via the dialog box, and hence, creates the operational environment for running his subroutine. The idea behind an environment generator is to allow the programmer to run and test program components without forcing him to go to great lengths to create a run-time environment. Because time taken writing a test harness takes away from application programming time, such a device would increase programmer productivity. However, since an environment generator is not inherently a visually oriented device (it could be implemented in a linguistically oriented system), we will not attempt to cover the subject further in the course of this thesis. Let us simply say that the visual interface would provide good tools for its implementation.

Other valuable tools to aid in writing and editing the program include a sophisticated, visually oriented text processor that simplifies program editing (one that includes cut, copy, and paste routines) and a program library, which allows the programmer to save program segments for future use; this is especially valuable during debugging because changes to the code can be made non-destructively--the original code can be saved when the corrective code is inserted, and thus, can be retrieved if the corrective code doesn't work out.

generator is probably not technologically feasible on a low-cost machine at this writing. The environment generator discussed in reference to our hypothetical program is useful because it simplifies the creation of program stubs and test harnesses--the degree to which this is possible depends on many factors, most notably the language used, and will not be discussed further in this thesis.

IV. THE USER INTERFACE

A. BACKGROUND

One of the major tasks facing the applications programmer is the creation of the user interface. The user interface is often the key to whether an application is successful as a product.

The user interface can take many forms, but it always serves the same function--it is the user's sole means of communicating with the computer. Hence, it is important to develop an interface which is both powerful and easy to use. Unfortunately, these two features are often diametrically opposed, since powerful implies more features, and more features implies greater complexity. For these reasons, much of the effort in developing software applications goes towards designing an elegant, application-specific interface.

B. THE CHANGING USER

The widespread use of computers today, in just about all areas of human endeavor, has necessitated a change in user interface design goals. Instead of creating applications only for computer-knowledgable clients, applications programmers must also create applications that can be learned by people with very little knowledge of computers or computer languages (the goal is to create applications that are powerful enough to satisfy the expert user while making them easy to learn and use). Thus, the programmer is faced with developing a user interface which is very easy to learn to use

(this is to be distinguished from easy to use) if he wants to sell his software product. All the power in the world is of no use to the client if he is unable to tap it. In addition, since the cost of computer time is no longer as significant as it used to be, the cost of human time has become very significant. Training personnel to use a computer or computer system is expensive. Training implies: 1) people who will do the training; 2) hours, days or weeks to be trained, on salary; and 3) low initial production from the person while the training sinks in and they become adept at using the system. Hence, there is a real market for easy-to-learn software applications.

The mention of a distinction between easy-to-learn and easy-to-use has been made--let's elaborate on this distinction. An application that is easy-to-learn provides an interface that is friendly to the user--he can quickly grasp (and remember) the steps necessary to accomplish his task. An application that is easy-to-use goes beyond this--it is elegant--it provides enormous power in few steps, flexibility (all users were not created equal), and design integrity--all of which contribute to the ease and pleasure with which it's used.

1. Managers And Casual Users

There are two other reasons that the market is growing for easy-to-learn and easy-to-use software: managers and casual users. The principal users of computers have, in the past, largely been technicians who have little need of easy-to-learn software because they are daily users of few applications. One new group of users emerging is the managers. Managers are more expensive to train, have less time to learn, and are more likely to need a variety of applications to suit their needs. The other new

group is composed of the casual users. These are users who often don't touch their computer for days or weeks at a time, plenty of time to forget everything they know about command language and syntax. Easy-to-learn and easy-to-use software is a must for this group if they are going to be productive.

C. THE STANDARDIZED INTERFACE

A standardized interface is one which varies little from application to application. There are two environments in which this standardization can take place: 1) the administrative environment; and 2) the environment of the application.

Standardization of the administrative environment is the first step in creating easy-to-learn and easy-to-use software. Such standardization necessitates that the user need learn only once how to perform the administrative tasks (creating, copying, transferring, and deleting files, erasing disks etc.)--this knowledge allows him to perform these tasks henceforth without regard for the application, since the tasks are performed in the same manner in all applications that adhere to the standardization (each application actually gives the user practice performing administrative tasks).

There are arguments both for and against a standardized interface. One of the disadvantages of it is that it prevents the programmer from tailoring the interface to the application. With all the myriads of applications, there are bound to be some that would benefit from having their own unique user interface. Another of the disadvantages of standardization is that it stifles creativity: the programmer may know exactly how he wants his program to

communicate to the user (perhaps far superior to the standard), but is prevented from implementing it because of the requirement for standardization. We will see how these problems can be partially resolved.

The advantages of standardization are primarily: 1) a standardized interface greatly simplifies the learning of more than one application; 2) a standardized interface significantly reduces development costs--less time is spent on the development of the user interface; and 3) a standardized interface implies tools that are provided to the programmer for the creation of that interface (toolbox routines)--these tools make the accomplishment of the simple things even simpler.

It is possible to create a standardized non-visual interface--many of these exist. They are usually operating system interfaces that are machine or brand specific (found on the same type of machine or on machines from the same manufacturer). Such interfaces are usually very simple and of limited power. It is also possible to create a menu-driven interface for use on machines with limited graphics capabilities (non-bit-mapped, low resolution machines). These interfaces are usually application specific and share a similar feature with visual interfaces in that they offer the user a selection of options from which to choose, the choice of which constitutes a command which activates a process or displays another menu. Menu driven interfaces are adequate for simple applications but do not lend themselves to standardization (since the menus are necessarily application specific) and are therefore of little help to the modern user.

Because of the need for creating easy-to-learn and easy-to-use applications, the natural choice for an interface is one that is visually oriented. The goal of such an interface is to create an environment where

the user feels comfortable--where he is presented with familiar objects no matter what the application (he uses a standard set of visual tools i.e. pointer, windows, desktop etc), and where his commands bring forth the desired, and most importantly, the expected results (the basic control actions required of the user are executed in a standard manner, from application to application).

The creation of the visual interface by the programmer can be made simpler by the use of existent toolbox routines that the applications programmer can call at will. The idea here is to make the programming tasks that should be simple, even simpler. The implementation of the functions that the application is to provide should be the highest priority of the programmer and the most difficult to do; the user interface should be easy by comparison. Unfortunately, this is not always the case. Without toolbox routines to aid in creating the visual interface, such a task can be formidable. This is due to the nature of the machines that run software utilizing a visual interface--they are bit-mapped and object oriented.

1. The Toolbox

The creation of a window or dialog box from scratch each time one was needed would just not be practical--either one would require a major programming effort. Of course, it is possible to standardize the visual interface without using toolbox routines: a software company could keep a library of such routines that it had already created for the use of its programmers (however, standardization among companies would be a major problem); or, in a worst case scenario, a general guideline could be agreed upon by software developers, whose programmers would find their own

unique ways of conforming to the standard. Two good methods of standardizing are:

- (a) Developing and distributing a machine specific ROM-based toolbox, so all the programmers would have the same tools.
- (b) Developing and distributing a software-based toolbox (this would be not be machine specific but would have the disadvantage of being slower than a ROM-based toolbox and would take up RAM and disk space.

No matter which method is employed, the use of such a toolbox can: 1) significantly reduce program development time and cost--programmers can spend most of their time on implementing the functional aspects of the program instead of the interface; and 2) help make the product easy-to-learn and easy-to-use by the user--the use of a toolbox leads to a great degree of standardization in the interface, thus reducing the number of application specific details of operation that the user must remember.

We might now ask the question, What tools should the toolbox contain? Since we are concerned with the standardized visual interface, we shall concentrate on the tools which aid the programmer in its creation (there are many other tools which could be included in such a toolbox to make the programmer's job easier in other areas of writing the program (i.e. floating point routines etc.).

First, let's make a list of the tasks the programmer might want to accomplish (these are the basic tasks involved in creating a visual interface):

- (a) Create the desktop environment.

- (b) Create windows that can be moved, overlayed, resized, scrolled through, opened and closed, selected and deselected, named, and which are capable of containing other objects.
- (c) Create dialog boxes that appear when needed, communicate a message to the user, enable a means of selecting options or typing a response, act on his response, and disappear when not needed.
- (d) Create icons that can be moved, opened, selected and deselected, and which can contain other icons through the use of windows (disk icons can be opened to a window which contains icons representing the next hierarchy level of objects; these icons can, in turn, be opened to windows which contain the next level of the hierarchy).
- (e) Create the environment(s) of the application (this would require drawing tools as well as tools enabling selection and the launching of processes).

There are two paths one can follow in developing the toolbox: you can make the tools high level, thus restricting the degree to which the interface can be tailored to the application (and restricting the programmers creativity), or you can make the tools low level, giving the programmer much greater flexibility in conforming to the interface but making it far more work for him to create it.

Use of a toolbox that is composed of a few high-level tools will result in programs whose interfaces are very similar, but which do not take full advantage of the visual aspect of the interface. The advantages of such a toolbox are: it restricts the number of different interfaces that can be created, and thus requires less time for the programmer to design the interface, and, because the tools are high level, it provides the simplest and fastest means of creating the visual interface.

Use of a toolbox that is composed of a great number of lower level tools enables programmers to create a great variety of visual features, allowing them to finely tailor the interface to the application (this takes full advantage of the power of the visual interface because the program's visual form follows its function, making it easy to learn and easy to use). There are two problems associated with this type of toolbox: 1) the great flexibility it gives the programmer in creating the application's interface can lead to the creation of one that is overly complex; and 2) the low level of the tools somewhat defeats the purpose of the toolbox, which was to simplify the creation of the visual interface.

We might ask what benefits the application user gets from the use of a standardized visual interface? First of all, it's important that we examine the nature of the environment that is created by this visual interface. Let's assume that the user communicates with the computer through the use of the keyboard and the mouse. The keyboard provides him with capability to enter data (or in some cases, commands) into the computer. Thus, the visual interface does little for the user in the entering of data (except dialog boxes, which are often used for data entry). The mouse, on the other hand, is almost solely devoted to working with the visual interface--it is the tool that makes such an interface work. The mouse is used for many tasks, but these can be categorized into two main areas: *control* and *non-control*.

D. DISCUSSION

Some distinction between control tasks and non-control tasks should be made here. Let us define *control tasks* as those which transfer control

from one part of the program to another. These are equivalent to commands entered via the keyboard on linguistically oriented systems. Control tasks include: launching and quitting applications, opening files or windows, selecting menu items in pull-down menus and desk accessories, and selecting command options within dialog boxes (in almost all other cases, the act of selection is a non-control task). All these control tasks activate some process, transfer control to that process, and manifest themselves in the form of screen changes at the time of activation. In a linguistically oriented interface, where the user is often presented with a standard prompt, these screen changes often take the form of hyphens or other symbols that appear in place of the prompt, telling the user that the system is acting on his command. In a visually oriented interface, the screen changes that occur take many forms, often logically related to the task being carried out. Some examples of such screen changes are: 1) double-clicking on the icon of a disk or folder causes a window to zoom (expand rapidly) out of the icon and be displayed on the screen showing its contents; 2) clicking the *close* box (a standard element of any window) of a window causes the window to implode back into the icon; and 3) selecting an item from a pull-down menu causes the selection to momentarily blink rapidly on and off and then to vanish, in addition to the screen changes caused by the execution of the command.

Non-control tasks are mainly selection oriented--their primary function is the selection of those objects to be operated on. This process is similar to prefix notation in arithmetic--first the items subject to the operation are chosen, and then the operation itself. Non-control tasks, like control tasks, provide instant feedback to the user as to whether they are

being carried out, but this feedback usually takes a different form. The screen changes are not as dramatic as with those associated with control tasks--generally, only the area of the screen where the pointer is affected. These changes usually involve a color (i.e. light to dark) or pattern change in the object being selected or deselected such that the object stands out on its background. An analogy can be made between control and non-control tasks and arithmetic operators and operands: non-control tasks are those that choose the operands; control tasks are those that choose the operators and activate them.

Undoubtedly, the single most important capability provided by the tools of the visual interface is that of selection. Almost all of the tools of the interface are accessible only through the use of the mouse, and almost all uses of the mouse involve the process of selection (this process is found in both control and non-control tasks and takes many forms).

Before going on, it is important to go over the basic procedures for using the mouse. The mouse is a device which, when moved across the surface of a worktable or desk, correspondingly moves a pointer across the computer screen. The mouse is equipped with a button which can be clicked (pressed once), double clicked (pressed twice rapidly in succession) or held down (this facilitates dragging--pointing, holding down the button, and moving the pointer). Each of these actions is associated with the performance of certain specific tasks:

- (a) *Pointing*, which is done by moving the mouse and thus the pointer on the screen to point at something.
- (b) *Selecting*, which is done by pointing and clicking when selecting objects, by pointing, clicking and dragging when selecting text, and

by pointing, dragging and releasing for menu item selection.

- (c) *Opening*, which is done by pointing and double clicking on an object or by selecting "Open" from a pull-down menu.
- (d) *Manipulating* which is actually moving things around, done by selecting and dragging the object to be moved. In developing a standardized visual interface, it is important that these basic methods of using the mouse are adhered to, since changing them would serve to confound and confuse the user.

Let's examine some of the tasks that the user is able to accomplish with the mouse. First, let's assume that upon entering the system, the user is placed in the environment of the desktop--the administrative environment of the computer. On this desktop he sees pull-down menus and icons. The pull-down menus list the set of commands available to him and the icons represent the data disks presently accessible, and perhaps a trashcan. The mouse provides the means to activate commands (control tasks) and also the means by which to manipulate objects (non-control tasks) on the screen. The menus on the desktop list the administrative functions that are available to the user (actually, they list all the commands that can be activated from the desktop, but only the ones highlighted are presently available for selection). Confronted with this desktop, the user can rearrange the icons on the screen or pull-down and survey menu items (non-control tasks accomplished by pointing and dragging). Selecting an item from a menu is a control task, which transfers control to some predefined process. Other things can be done by the user also: by moving one disk icon on top of another he can copy the contents of the first disk to the other; by double-clicking on any icon he can open it to see what files it contains; by clicking on an icon he can select that icon for some process to be activated via the menus.

Opening a disk icon, the user is confronted with a window containing the icons of all the files on that disk. These icons are application-specific--their appearance conveys information about the application that created them. Double-clicking on a file will launch the application that created that file (the application will then open to that file), while double-clicking on the icon of an application will launch that application, which will open to a new file. Once inside an application, the user is presented with a background that is specific to the application; however, he retains the visual tools that he started with. He still has pull-down menus and/or icons, and the basic operations of the mouse in pointing, selecting, opening and dragging are the same. Naturally, every application has different requirements regarding functions and features but it is important to maintain the integrity of the *logic* of the visual tools (for instance, having the user double-click on an icon to "select" it would go against the logic of the tools of the visual interface).

What advantages does such a visual interface have over a linguistically based one? To begin with, a visual interface of this sort eliminates command syntax errors. On traditional systems, where all commands are entered via the keyboard, the user is required to enter commands in a specific format--altering this format either generates an error message or executes the wrong command. This simply does not occur on with a visual interface. The only commands that will be accepted by the system are those that are highlighted in the pull-down menus--the user simply cannot activate commands that are inappropriate to the environment which he is in (they are not highlighted and will not respond to selection), nor can he execute a command that doesn't exist (since he can only select commands

available onscreen, which by definition are part of the application's command language).

The elimination of command syntax and static semantic errors saves the user time--no doubt about it--but there is another area related to this that is even more beneficial: the visual interface relieves the user of the burden of having to memorize the command language of the software. This is one of the most powerful features of the visual interface and its importance in facilitating ease of use cannot be overstated. In present day linguistically oriented interfaces, the users sole means of accomplishing control tasks is by typing commands at the keyboard. An application which contains more than a few features will undoubtedly require an extensive command language. Learning and maintaining a knowledge of such a language is only made possible by the very frequent use of the application--even then, features which are not used very often might necessitate looking up the desired command in a manual. Thus, casual users are at a great disadvantage with a linguistically oriented interface, because they simply don't use the application enough to remember its command language. Managers and other users who frequently use more than one application are put at an additional disadvantage: memorizing several different command languages is an enormous task and can lead to confusion (one application's commands might be confused with those of another application).

With a visual interface, the command language (of the application) is built into the application in the form of choices offered on pull-down menus, dialog boxes, and desk accessories. This is in addition to the standardized command language of the interface, which consists mainly of the basic functions of the mouse. The visual interface facilitates a dialog

between the user and the computer, as opposed to the monologue from the the user to the machine that is prevalent in linguistically based interfaces. Before going on, it is important to discuss this concept, which is a fundamental difference between the two interfaces.

Most linguistically oriented interfaces demand that the user be thoroughly familiar with the command language of the application, and assume that the user always knows what he wants to do next. This assumption, though true in many situations, may be false in others. Complex applications often require that the user go through a step-by-step process to reach a desired end. The steps of this process are not always obvious. Linguistically oriented interfaces might require the user to write down the series of commands needed to perform a desired function. Unless the user is wholly familiar with the application, he is never quite sure what commands are executable from a given point within the program--even if he is, if there are a number of them, he must be able to keep in mind what he is trying to do and know which one will accomplish his task. In this way, communication with such an interface is one-way--the user must know what commands to issue without any help from the computer.

The visual interface, with its command language built into the interface and the application, conducts a dialog with the user. Pull-down menus (showing the user a list of possible actions) and dialog boxes (actual queries to the user demanding a response) continually offer the user choices of what to do next; the user then makes his choice. This two-way communication serves to guide the user down the decision path--he is no longer forced to plot out the details of his actions beforehand or to memorize the commands that bring about those actions. If the user gets to

the point where he is not quite sure what to do next, he can pull down menus until he finds the desired command. It is this feature which is especially useful to the new user, the casual user, and the user of many applications, since it makes no demands on their memories, only on their decision making capabilities.

The fact that the command language is presented to the user via dialog boxes and pull-down menus and need not be memorized puts the user in the unusual position of being able to sit down at the computer, start the application, and work productively (as long as he has the knowledge needed to solve his particular problem--the interface will not do this for him) without ever having to glance at an instruction manual for the application. Naturally, there might be subtle nuances of the program that cannot be determined by such experimentation--for these one must still consult the manual--but the point is that the user can simply "figure out" the basic functions of the program by just diving in and using it. Such a thing would be unthinkable for most applications which have linguistically oriented interfaces--you cannot execute commands which you do not know exist. Let's briefly examine the reasons why a standardized visual interface makes this possible.

The difference between a standardized visual interface and a non-standardized visual interface is that, with the former, upon entering any application, the user is greeted by a familiar visual environment (generally the desktop) for which he knows the basic command language, whereas with the latter, he enters an unfamiliar environment in which he has no knowledge of the command language. The standardization of the visual interface is very important because it gives the user a solid

foundation from which to start the application: he knows the basic operations of his principal tool, the mouse (clicking, double clicking, dragging, pointing) and the capabilities those operations provide; in addition, he is familiar with the desktop environment and the administrative tasks that can be accomplished there. Thus, he already has some knowledge of the operation of the application. Upon entering the program environment, which may be quite different from the desktop environment, the user is still armed with his knowledge of the function of his basic tool, the mouse, and is able manipulate or select objects, or activate processes in the same manner that he is accustomed to. The nature of the application may require that he extend this knowledge--application specific functions may necessitate variations or additions to the capabilities provided by the mouse--but it should never require that he "un-learn" those basic functions. Note that it is crucial that applications follow the logic of the standardized visual interface if the advantages of the visual interface are to be exploited. Departing from this logic--requiring the user to conduct operations in ways dissimilar from those he has learned--forces the user to memorize a new command language (which is difficult because it departs from the *philosophy* of that which he has already learned), and thus constitutes a failure to utilize one of the most powerful features of the visual interface.

Following the *philosophy* of the standardized visual interface consists of the following:

- (a) Retaining the basic functions of the mouse--any additional functions or variations must be easy to remember--they must maintain logical

integrity with the basic functions (they should be easy to remember because they should become intuitively obvious once they are performed a few times).

- (b) Retaining the standard visual tools and their means of operation: windows, icons, dialog boxes, desktop, pull-down menus.
- (c) Providing visual tools specific to the application that are extensions of the standard visual tools and whose operation is logically similar.

Adherence by the application programmer to the philosophy of the interface makes it possible for the user to jump into an application with no prior knowledge of its operation, and to navigate through the program using just the standard tools of the interface. This is a powerful feature which is of great value to new users, casual users, and users of many applications.

Another advantage that the visual interface has over the linguistically oriented interface is the simplification of operations which would normally require a complex command sequence. To illustrate this, let's discuss one of the complaints often voiced about the visual interface: users of non-visual interfaces frequently complain that while the visual interface simplifies use of the computer for the novice, who doesn't always know where he's going, it complicates the use of the computer by the expert, who knows exactly where he is going. In other words, the tools of the visual interface can be tedious when the user knows what he is doing.

Before going on, we must ask ourselves the question: is the method of command activation really the complaint of these expert users or are they really complaining that the visual interface slows down the machine? The visual tools are powerful, but much of the computer power goes into the creation and management of the visual interface instead of being directed toward accomplishing the task at hand. When the user activates a command

to perform a certain function, the command sets many processes in motion which may have little to do with the actual function to be performed. These processes include: screen changes (to indicate execution); background changes (from the desktop environment to that of the application, or from one environment of the application to another); percent done indicators, which give the user some idea of how long an operation is going to take, allowing him to be productive elsewhere while he's waiting [Ref. 5]; pull-down menu modification or creation (different selections might be highlighted or new pull-down menus created); dialog box presentation; window and icon modification or creation (windows and icons might be created or highlighted/de-highlighted). It is not the intent of this thesis to go into the area of performance slowdowns due to the maintenance of the visual interface. Let it suffice to say that with technological advances and the use of separate processors to manage the visual interface, it is probably no longer a relevant issue.

Since the visual interface does tend to guide the user down the decision path, it would seem that more steps would be necessary to get from point A to point B. This is not necessarily true. The command language of an application that uses a visual interface is generally more powerful (one command in the visual interface is often equal to several in the non-visual interface) and more flexible (the visual interface often provides more than one way to go from point A to point B) than that of applications utilizing linguistically oriented interfaces (Note: as we shall see, there are inflexibilities associated with the visual interface as well). The visual tools are the source of this power. For instance, let's say that you wanted to copy a file from one disk to another. With the non-visual interface, you

would have to type a specific command which included the name of the file to be copied, the disk copied from, and the disk copied to. With the visual interface, you could select the file in the source disk window and then drag the selection to the destination disk window. Such an operation could be performed faster than that involved with the non-visual interface, with less chance of error--no chance of a command syntax error and little chance of a procedural error since the operation follows the logic of the user's thinking ("take this file on this disk and copy it onto this other disk"). The power of the visual tools translates the physical act of dragging an icon into a command sequence that accomplishes the copying of the file.

Although there are many examples like the one above, there is some truth in the argument made by the expert users--the argument that the use of the tools of the visual interface (i.e. pull-down menus etc.) is far more time consuming than the use of typed-in commands. This is a valid criticism because the visual interface usually requires that the user adhere to the somewhat inflexible rules of the environment. This inflexibility can necessitate otherwise unnecessary steps on the part of the user. Let's look at an example: the user has a number of windows onscreen in the desktop environment and wishes to transfer a file from one window to another (the user knows the names of the windows and the file)--unless the two windows involved in the transfer, and the file to be transferred, happen to be visible and recognizable, the user must shuffle windows around (bring them to front/send them to back, or close them) until they are visible, in order to accomplish the transfer. Selecting and dragging the icon of the file from the source window to the destination window will accomplish the transfer; forcing the user to manipulate the windows to get to this transfer

step is a waste of time. A typed-in transfer command containing the name of the source window, the file name, and the destination window would speed up this process considerably.

Obviously, if the user knows the command that he wants to execute, typing it in may often be faster than finding and selecting it from a pull-down menu, especially for a fast typist. However, there are ways of making the visual interface just as productive as the non-visual interface for these expert users:

- (a) Create keyboard commands that coincide with pull-down menu selections.
- (b) Create a redundant command language--a linguistic command language to lie beneath the visual command language.
- (c) Allow the user to create macros or executives.

The idea of an underlying linguistic command language is intriguing. It would solve many of our problems. First, it would satisfy the expert users--they could ignore the mouse (well, almost) and work entirely from the keyboard (this might not always be desirable, even for the experts, in extremely complex applications). The clumsiness of the visual tools in certain situations could be bypassed by these experienced users, thus, saving them time and adding power. Second, an underlying linguistic command language would simplify the creation and modification of user defined macros or executives (single commands that activate multiple commands), since it would break down the visual commands into lower-level, type-in commands that could be selectively edited. Third, such a language would in no way affect the normal visual interface--if desired,

the user need only use the visual tools via the mouse. Thus, the easy-to-learn, easy-to-use advantages of the standardized visual interface would still apply; only the expert user would want to learn the linguistic command language. Taking all these things into account, it is easy to see that the development of such a language could only benefit users (they would have "the best of both worlds"), as long as no compromises were made regarding the visual command language or the interface to facilitate that development.

The growth in the power of software applications generally entails a growth in the numbers of features they provide, since, the more features we have at our disposal, the more tasks we can accomplish. The greater the number of features, the more control decisions must be made to utilize those features. The number of these control decisions is what determines the complexity of the software. We'll define *complex applications* as those which require many control decisions to be made. Thus, by definition, as the level of complexity increases, the number of control decisions to be made increases.

Perhaps the greatest benefit in using a standardized visual interface is that it enables the user to use increasingly complex applications. Let's make a bold statement here and see if we can back it up: The more complex the application, the more necessary it is to have a visual interface.

Using a non-visual interface with a complex application, the user is subject to four side effects:

- (a) Regardless of his frequency of use or how expert he is, he simply cannot remember the entire command language.
- (b) Infrequently used features, of which there may be many, will

necessitate frequent referrals to the application manual.

- (c) Many of the features of the application will not be used.
- (d) The user may become hopelessly lost in the application, having no idea of what he should do next.

A visual interface is not as susceptible to these side effects. The command language of the application need not be learned or remembered (at least, not the specific commands themselves--the user still must know what the commands do), so the entire command language is always accessible to the user. This feature of the visual interface makes it invulnerable to two of the side effects encountered when using a complex application with a non-visual interface: infrequently used features can be utilized just as easily as frequently used features, thus, all of the features of the program are more likely to be used. Additionally, the user is less likely to become lost in the application since he always has a frame of reference--the screen background--and since he is guided by the application (highlighted choices on pull-down menus are the only ones appropriate for his next action) in the problem solving process.

When inside a complex application, the user has a number of questions to consider (whether the user is using a non-visual or a visual interface has a profound effect on the answers to these questions):

- (a) What is the problem he is trying to solve? The answer to this question will hopefully be easy; if not, the visual interface may help him by offering him choices of what he wants to do.
- (b) What is the sub-problem he is trying to solve? To answer this question, the user must have a clear idea of where he is in both the application and the problem solving process. With the non-visual interface, if the user is familiar with the application, is concentrating on what he is doing, and can pursue the problem

solving process uninterrupted (interruptions can cause him to lose his place in the problem solving process so that when he comes back to the machine he will not know where he is--and it may not be easy to find out), he will probably know what this sub-problem is. With the visual interface, the user is guided in the decision process by the application--he can only make choices within pull-down menus that are possible with respect to where he is in the application. The visual interface also makes it easier to come back from an interruption, since it provides a visual frame of reference.

- (c) What is the feature of the application that will enable him to solve this sub-problem? The non-visual interface requires that the user memorize all the features of the application--this can be an enormous task in a complex application. By contrast, the visual interface always displays all the features provided within a given frame of reference. The user does need to know what the features do however.
- (d) How can this feature be called upon--what commands will activate it? The answer to this depends on the extent of his knowledge about the application's command language and on how familiar he is with using this particular feature. If he is using a non-visual interface and it is not a familiar feature, it is unlikely that he will know the applicable command to activate it, forcing him to refer to the manual. The pull-down menu selections in the visual interface provide all the commands that can be executed from the environment he is in and thus can help him immensely.
- (e) In what order should commands be executed? The answer to this depends on his knowledge of the application and his knowledge of the problem solving process for his particular problem. The non-visual interface may force him to carefully plan and write down his intended steps ahead of time, in order to proceed smoothly towards the problem solution--this may involve a thorough study of the problem solving process and extensive referrals to the manual for the right commands to carry out the process. The visual interface, with its pull-down menus, requires that the user have a basic knowledge of how to solve the problem--enough so that he can recognize what step comes next when he sees the command to activate it. This is similar to a multiple choice test in academics,

pull-down menus merely require that the user be able to recognize the right command when he sees it, not that the user memorize all the choices, as with the non-visual interface. Also, the visual interface makes it easier to execute steps in the right order through the use of highlighting in the pull-down menus--it restricts the number of choices available to those pertinent to the present state of the problem solving process.

- (f) What is the next step towards solving the problem? The problem confronting the user when using a complex application is that the problem solving process is not necessarily a linear process--from any point in the process the user may be presented with numerous choices of where he wants to go next. Whether or not he is able to make a choice depends on his knowledge of where he is, and what task he wants to accomplish next, as well as his knowledge of the application itself. The non-visual interface gives him little indication of where he is--it provides no frame of reference; it provides little help in deciding what task he wants to accomplish next--it only displays a prompt symbol indicating it is waiting for his input. By contrast, the visual interface provides a visual frame of reference telling him exactly where he is and displays pull-down menus that can help him in deciding what to do next--if he has executed the correct commands up to this point, the choice of what to do next will always be among the highlighted selections in the pull-down menus.

V. CONCLUSION

The reader has been given some insight into current visual technology and how visual tools can be used to help application programmers and users. From the application programmer's standpoint, these visual tools take the form of sophisticated program processors which aid him in the writing, editing, debugging and testing of the code. Nevertheless, it is the richness of the visual interface that is of the greatest value to application programmers--it enables them to create an environment for their application that is tailored to the conceptual model of the user--an environment that facilitates fast learning and ease of use. The toolbox is the key to creating this environment. Its use saves the programmer from investing enormous efforts in the creation of the visual interface and promotes standardization among applications. In addition, the toolbox helps the programmer to create an environment that is unique to the application.

From the application users point of view, the visual interface offers a friendly and familiar environment; applications which follow a standardized visual interface are easy to learn and easy to use. As applications grow in complexity and number, we have observed with the non-visual interface that there is a limiting factor regarding the number and complexity of applications that a single user can use effectively. This does not appear to be true for the visual interface--as long as it follows the conceptual model of the user, its standardized administrative environment and adherence to the logic of the functions of its tools make it effective in dealing with multiple, complex applications.

LIST OF REFERENCES

1. MacLennan, B. J., *Principles of Programming Languages*, p. 454, Holt, Rinehart and Winston, 1983.
2. Hunter, J. E., *The Formal Specification Of A Visual Display Device. Design And Implementation*, M.S. Thesis, Naval Postgraduate School, Monterey, CA, June 1985.
3. Raeder, G., "A Survey of Current Graphical Programming Techniques", *Computer*, Volume 18, No. 8, p. 11-25, August 1985.
4. Teitelbaum, T., and Reps, T., "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment", *Communications of the ACM*, Volume 24, No. 9, September 1981.
5. Moriconi, M., and Hare, D. F., "Visualizing Program Designs Through PegaSys", *Computer*, Volume 18, No. 8, p. 72-85, August 1985.
6. Myers, B. A., "The Importance of Percent-Done Progress Indicators for Computer-Human Interfaces", *CHI '85 Proceedings*, p. 11-17, April 1985.

INITIAL DISTRIBUTION LIST

	No. Copies
1. Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6045	2
2. Superintendent Attn: Library (Code 0142) Naval Postgraduate School Monterey, California 93943-5002	2
3. Chairman (Code 52) Department of Computer Science Naval Postgraduate School Monterey, California 93943-5000	1
4. Curricular Officer (Code 33) Weapons Engineering Naval Postgraduate School Monterey, California, 93943-5000	1
5. Lt. Michael N. Fredericksen Department Head School Class 93 Surface Warfare Officers Command Newport, R.I.	2
6. Gordon H. Bradley Code 52 BZ Naval Postgraduate School Monterey, California 93943-5000	2

END

DTIC

6-86